Conditional Estimation of Vector Patterns in Remote Sensing and GIS

Interim Report 5

Principal investigator: Dr. J.M.F. Masuch

1 May 1999

United States Army
European Research Office of the U.S. Army

USARDSG-UK, Edison House
223 Old Marylebone Road
London, NW1 5TH
England

Contract Number: N68171-97 C 9027

CCSOM/Applied Logic Laboratory
PSCW - Universiteit van Amsterdam
Sarphatistraat 143
1018 GD AMSTERDAM
The Netherlands
tel: #.31.20.525 28 52
fax: #.31.20.525 28 00
e-mail: ccsoff@ccsom.uva.nl

R&D 8249-EN-01
Broad Area Announcement Proposal
submitted to the
Remote Sensing / GIS Center USACRREL
72 Lyme Road
Hanover, New Hampshire 03755 USA

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 27,1999	5th interim report (07 May)

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Conditional Estimation of Vector Patterns in Remote Sensing and GIS	N68171 97 C 9027
6. AUTHOR(S) Dr.J.M.F.Masuch	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
CCSOM / PSCW / University of Amsterdam Sarphatistraat 143 1018 GD AMSTERDAM NL	BAA III 99-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

19990628 099

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

We examine minimum sufficient data requirements and digital compression methods for processing vector polygonal data using X11R6 libraries. Algorithms are developed for the fast import, export, and compaction of binary data using standard graphical primitives. Algorithms are shown to be adaptable to both raster and vector image processing depending on *grafport* conditions and the level of vector modeling required for pattern identification. The concepts of minimum sufficient data, data classification, and feature extraction are reviewed to understand how vector architectures compare with other data translation and image integration methods. The discussion focuses on the efficient conversion of raster images to vector equivalent models for use within Computer Aided Design (CAD) and Geographic Information Systems (GIS). All algorithms build upon prior research efforts outlined within the cumulative ERO research program. For example, data import and data export algorithms build upon previous functions and procedures developed for the fast import and export of raster data using traditional image processing techniques. Algorithms include raster and vector fields conducive to point, line, arc, and polygon geometric models. The effort is directed toward device independent computer architectures as a means to support common UNIX platforms. All data structures employ Open Systems Foundation (OSF) C/C++ language techniques for direct application across heterogeneous networks.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Remote Sensing, Statistics, Artificial Intelligence, Neural Networks	30
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

1. Title

Conditional Estimation of Vector Patterns in Remote Sensing and GIS:
Interim Report 5

2. Abstract

We examine minimum sufficient data requirements and digital compression methods for processing vector polygonal data using X11R6 libraries. Algorithms are developed for the fast import, export, and compaction of binary data using standard graphical primitives. Algorithms are shown to be adaptable to both raster and vector image processing depending on *grafport* conditions and the level of vector modeling required for pattern identification. The concepts of minimum sufficient data, data classification, and feature extraction are reviewed to understand how vector architectures compare with other data translation and image integration methods. The discussion focuses on the efficient conversion of raster images to vector equivalent models for use within Computer Aided Design (CAD) and Geographic Information Systems (GIS). All algorithms build upon prior research efforts outlined within the cumulative ERO research program. For example, data import and data export algorithms build upon previous functions and procedures developed for the fast import and export of raster data using traditional image processing techniques. Algorithms include raster and vector fields conducive to point, line, arc, and polygon geometric models. The effort is directed toward device independent computer architectures as a means to support common UNIX platforms. All data structures employ Open Systems Foundation (OSF) C/C++ language techniques for direct application across heterogeneous networks.

3. Introduction

Compressed data structures are employed as a method to store and retrieve raster and vector patterns acquired from Image Processing (IP), Geographic Information Systems (GIS), and Computer Aided Design (CAD) applications. Algorithms use Segmentation and Run-Length Encoding (RLE) techniques to compress individual data fields contained within an acquired raster image. Segmentation is also used to display and retrieve information using pointer arithmetic that is efficiently implemented within the C/C++ language (Clark et. al., 1991).

Segmentation algorithms employ a piecewise functional approximation (Anderson [1994] and Baase [1992]) to represent all portrayed data. Image data are fit according to an error criterion with line (or polynomial) segments. The output from the algorithm is a string of triples $\{(x_i,y_i), A_i, B_i\}$; $i = 1,.., \Theta_s$, where Θ_s is the number of segments, $y = A_i x + B_i$ is the linear approximation to the ith data segment, and (x_i,y_i) is the right endpoint of the line segment. This string is translated into the terminal symbols (tokens) of a grammar under the control of parameters appropriate to the application.

The structural analysis is accomplished by a left-to-right parser for a grammar that defines more complex relationships among the terminal symbols. In this context, line segments form a radiometric peak or geometric pattern within the raster image and a string of triples is used to represent all geometric patterns with distinct inflection points. Using piecewise interpolation, polygons are decomposed into distinct triples $\{(x_i,y_i), A_i, B_i\}$, and the entire pattern is compressed for storage within a common file.

The advantages of this approach include: processing speed, generality, and the mathematical tractability that approximation theory provides. However, the approach is more one of numerical analysis rather than pattern analysis. This comment also applies to proposals to use truncated K-L series expansions in pattern analysis (Baum [1989], Chandrasekaran and Harley [1993]). Such preprocessing usually results in arbitrary segmentations and requires excessive time for scanning and matching of all the data. Furthermore, the segments extracted may not be meaningful in the context of a specific application. In separating the analysis of structure from the extraction of morphs, each process is excluded from information available to the other. That is, the extraction of individual morphs must proceed in ignorance of the a priori combinatorial restrictions known by the structural component, and the structural component cannot profit from the intermediate work of extraction. Integrated segmentation is exemplified by the work of Clark [1990, 1991] for hyperspectral waveforms and Karp [1986] for vector pattern recognition. The salient points of the integrated approach include: (1) Knowledge Based Segmentation and (2) Parsed Segmentation.

4. Knowledge Based Segmentation

Within knowledge based segmentation, there is an a priori (Bayesian) knowledge of the possible segments and/or geometric patterns contained within the raster image. For example, in a signature library, it is known that the majority of spectral patterns include an upslope, a trailing edge, and a series of convex and/or concave patterns. Morphs can be defined that functionally represent the general shape and distribution of the signature data (e.g. the functional representation may include measures of complexity from local extrema to exponential segments). The morphs represent a reduced set of instructions that can be quickly applied to the original image to summarize and reconstruct all information.

Within the context of spectral decomposition, Boardman [1990, 1992], discusses the desirability of knowledge-based search for distinguishing features in preference to scanning the entire scene with low-level operators. The term "knowledge-based," popular in the artificial intelligence (AI) literature, generally refers to "non-statistical" a priori information, although statistical information and Bayes' theorem are also acceptable in AI (Boardman [1993], Baum [1989], Kirkpatrick [1983]). Using AI methods, prior knowledge is represented by means of decision trees (Kruskal [1993]), graph models and decision graphs (Gibbons [1990]), grammars (Hornik et. al. [1989]), and neural decision trees (Weigend et. al. [1990]).

5. Parsed Segmentation

Within parsed segmentation, scanning is "bottom-up" and "top-down" and not restricted to "left-to-right". In other words, the more prominent morphs are sought first regardless of their location, and then the grammar is used to predict where other morphs are to be found. In the case of spectral decomposition, morphs are defined in order of significance corresponding to magnitude and duration of a given waveform. Having extracted the signature, a grammar (function or procedure) is used to predict the next morph to be scanned (e.g. the trailing edge of a given peak). Having extracted the trailing edge, the next morphs to be scanned might be peaks and a diacritic notch between the upslope and the trailing edge. In this fashion, the extracted morphs clue the system to the rest of the structures to be searched and also allow future searches to be performed over restricted intervals (e.g. Mazer et. al. [1988]).

Parsed segmentation methods are commonly used to construct formal parse trees. In Gibbons [1990], parse trees are "seeded" by first scanning the entire input for prominent features (usually pixels with similar radiometric levels). Features are sought and *Partial Parse Trees (PPT)* are assembled anywhere within the raster image. The PPT represents the grammatical structure, while the terminals of the tree are words in the data believed to exist within the vector image. PPT's are enlarged by using the grammar to guide the search for features within the neighborhood of "known" vector patterns and to connect several PPT's into one. Analysis can terminate any time the complete neighborhood is "covered" by some PPT. The bottom-up non-left-right approach is attractive because of the ability to search first for reliable morphs regardless of within image location and to guide future analysis.

There is an analogous technique employed when decision trees rather than grammars are used to represent the possible structures. For example, in the analysis of Synthetic Aperture Radar (SAR) data, one searches for each inflection with respect to those features already extracted from the total image. If processing leads to an impossible structural outcome, the analysis is backed up and the extraction of morphs begins within a new neighborhood inside the SAR image.

In Clark [1991], isolation of a given morph narrows the structural possibilities, while the present state of structural possibilities dictates the next morph extraction to be attempted. The parsing approach can also be top-down and left-to-right (Yuhas et. al [1992]). By going top-down only syntactically and semantically acceptable configurations are considered, and very expensive preprocessing can be minimized. By going left-to-right, classical parsing methods can be used and the state of analysis is easily recorded. One can define a probability or error fit that a given morph matches a given segment of raw data. This probability can also be backed up a PPT to yield a figure of merit (ranked likelihood measure) for a partial parse. The probability of a PPT can be changed by "semantic conditioning" (Hornik [1989], Kruskal [1993]). By maintaining this figure of merit for all partial analyses only the most promising routes are extended, or in the case of complete analyses only the most likely cases need to be accepted.

Plex grammars (Beakley and Tuteur [1992]) can be employed within VECTOR imagery if proper data structures are defined prior to image import and export. Plex grammars involve primitive entities called *napes*. Each nape has a finite number of attaching points, each of which has an associated identifier. Napes are combined by bringing attaching points into coincidence. A picture description language (Akl [1985]) can be used to describe pictorial patterns, the primitive elements of which have arbitrary shapes and distinguished heads and tails. The hierarchic structure of a picture is defined by using a "picture description grammar" to combine expressions in the picture description language. Baase [1992] presents an algorithm for representing pictures in terms of trees. A tree grammar that generates decision nodes is thus a formal description of the corresponding set of patterns.

There are some problems in using trees and tree grammars in the manner of Baase. First, since trees are acyclic graphs, a single tree cannot completely describe the connectivity of a closed figure. Second, trees introduce ambiguity into a pattern that may not itself be ambiguous. This ambiguity arises because the description of a figure by means of a tree requires a segmentation of the figure described, and an ordering of the segments. A different choice of segments and ordering would result in a different description of the pattern. Graphs and graph grammars are probably more appropriate structures for describing line drawings because cyclic graphs can completely describe the connectivity of closed figures, and a graph description need not order the parts of a figure. As a result, it has often been suggested that transformation rules might be just as useful in pattern analysis as they have been in providing insights into the structure of natural languages.

Parsed segmentation can be enhanced using transformation grammars to parsimoniously describe the salient features within the image. A transformation grammar is defined as $G_\phi = (G,\phi)$, where G is a reasonably simple "base" grammar such as a context-free grammar, and ϕ is a mapping that maps a structure in G, i.e., a tree, into a related tree. Both Karp [1986] and Boardman [1989] present detailed examples of a transformation grammar derivation for a class of polygonal patterns. The examples illustrate how a context-free base grammar and transformation rules for deletion of the interior lines of the generated patterns lead to much simpler derivation than a more direct approach involving a context-sensitive grammar. Chandrasekaran and Harley [1993] also discuss the application of transformation rules to trees generated by tree grammars. Their paper considers transformations to: i) duplicate patterns, i.e., to represent complex patterns as a periodic repetition of some simple pattern, and ii) relate two occurrences of the same pattern, including one that has undergone a shape-preserving transformation such as rotation, translation, or reflection. In Boardman [1992], it is suggested that the syntactic structure of transform grammars be described by context-free web grammars or array grammars. However, there are some relationships between structure diagrams that cannot be described by context-free web grammars, for example, the relationship between equivalent structural formulas of the same pattern or attribute. Transformation rules can be used to transform equivalent diagrams into a canonical form, as well as to combine diagrams and to decompose diagrams into "kernel" diagrams. These are the ideas motivating the work of Clark et. al [1991], who introduced the concept of a transformation web grammar. Because of the potential of graphs for describing patterns of practical interest, graph and transformation graph grammars are likely to receive increasing attention in syntactic pattern recognition.

6. Algorithms for Efficient Data Vector Conversion

Using standard C/C++ data structures, parsed segmentation methods may be employed to manage the import and export of vector image data. Parsed segmentation requires detailed management of information using indexes. The indexes correspond to either pointer or handle data structures within the C/C++ language. An example parsed segmentation algorithm is shown in Figure (1). In this example, a simple function (Add_VECTOR) has been constructed to add all vector patterns contained within two separate parsed objects: VECTOR_1 and VECTOR_2. The function returns a concatenated data handle within the VECTOR_Handle data structure. Note that this function uses "double-inflection" to manage information by dynamically linking data across handles. As data is spooled into a particular data handle, it is referenced, and then concatenated with a second data handle to produce a single composite result. In Figure (1) and throughout this discussion, the ValidHandle function is applied to determine the fidelity of the information. In positions (i-iii) of Figure (1), data is rejected when the validity of the data handle cannot be determined. In this manner, all

```
\*********************************************************************/

VECTOR_Handle Add_VECTOR (VECTOR_1,VECTOR_2)
      VECTOR_Handle     VECTOR_1, VECTOR_2;
{
            VECTOR_Handle     the_VECTOR;
            Rect         newFrame;

      /* If one of the _VECTOR_s is no good, return only the good one,
if any...    */
      if (!ValidHandle((Handle)VECTOR_1) ||
!ValidHandle((Handle)VECTOR_2)) {
      Alert("\PAt Least One Image Handle ","\PIs Bad or Mangled ",
"\PTry Operation Again! ","\P");

            if (ValidHandle((Handle)_VECTOR_1))
      (i)
                  return(VECTOR_1);
            else if (ValidHandle((Handle)VECTOR_2))
      (ii)
                  return(VECTOR_2);
            else
                  return(NULL);
      (iii)
                  /* NULL if neither _VECTOR_Handle is valid      */
      }

      /* Find frame for new VECTOR: must be large to hold both V_s..*/
      UnionRect(&((**VECTOR_1)VECTOR_Frame),
                  &((**VECTOR_2)VECTOR_Frame),&newFrame);
      (iv)

            the_VECTOR_ = Open_VECTOR (&newFrame);
      (v)
            Draw_VECTOR(_VECTOR_1,& ((**_VECTOR_1).VECTOR_Frame));
            (vi)
            Draw_VECTOR(_VECTOR_2,& ((**_VECTOR_2).VECTOR_Frame));
            (vii)
            Close_VECTOR();
      (viii)

      return(the_VECTOR);
};
```

Figure (1): Function Add_VECTOR. The function adds two VECTOR handles and returns a parsed VECTOR_Handle. Error checking is managed using the ValidHandle function which tests for valid address space on the host platform.

```
\*********************************************************************/
```

erroneous information is rejected prior to concatenation within the Add_VECTOR function. In position (iv), the procedure UnionRect is used to combine the geometry of the two data handles. Using this technique the concatenated data handle contains a single minimum sufficient rectangle that is the union of the two rectangles concatenated within VECTOR_1 and VECTOR_2. The procedure Open_VECTOR is used to define a new data handle. The Open_VECTOR procedure receives information between position (v) and position (viii). The Draw_VECTOR procedure places the graphical description of the vector patterns within the open data handle. Note that a separate Draw_VECTOR procedure is required for each data handle. As a result, two separate Draw_VECTOR commands are shown in Figure (1) at position (vi) and position (vii) respectively. The Add_VECTOR function illustrates the general pointer and handle arithmetic that is required to support parsed segmentation of VECTOR data. As shown, both geometry and data must be managed within each function and procedure to ensure that all information is accurately placed within the composite result.

In Figure (2), separate VECTOR patterns are merged to form a single co-registered overlay. The function Over_VECTOR merges the digital information, calculates the geometric Union of the digital data, and then overlays the final product. As applied within Figure (1), the ValidHandle function is used to check all digital information prior to physical overlay of the data. The validity check is accomplished in position (i-iii). In position (iv), the Union (minimum sufficient rectangle) is calculated for the overlay. Note that the Union rectangle will always be sufficiently large to contain the new product overlay but will never exceed the bounds of the original data. In position (v), the procedure VECTOR_ToBits is applied to move the digital information onto an off-screen grafport. This procedure allows the user to store information in a single location that is not directly visible to the user. As in Figure (1), the Open_VECTOR and the Close_VECTOR functions are applied to the separate data handles to merge all digital information. The actual VECTOR data (contained in VECTOR_1 and VECTOR_2) is combined using the Draw_VECTOR procedure in conjunction with the standard CopyBits utility. In position (vi), the overlay data handle is opened and set to receive the new digital information (resulting from the overlay process). The overlay begins with the initial Draw_VECTOR procedure in position (vii). The CopyBits utility (position viii) moves the base image (i.e. the background

```
\*********************************************************************/

VECTOR Handle Over_VECTOR (VECTOR_1,_VECTOR_2,thePen)
      VECTOR_Handle       VECTOR_1, VECTOR_2;
      int                 thePen;


{
            VECTOR_Handle      the_VECTOR;
            Rect         newFrame;
            BitMap             theBits;
            GrafPtr            drawPort;

      /* If one of the VECTOR's is no good, return only the good one, if
any...        */
      if (!ValidHandle((Handle) VECTOR_1) || !ValidHandle((Handle)
VECTOR_2)) {
      Alert("\PAt Least One Image Handle ","\PIs Bad or Mangled ",
"\PTry Operation Again! ","\P");

            if (ValidHandle((Handle)VECTOR_1))
      (i)
                  return(VECTOR_1);
            else if (ValidHandle((Handle)VECTOR_2))
      (ii)
                  return(VECTOR_2);
            else
                  return(NULL);
      (iii)
            /* Return NULL if neither _VECTOR_Handle is valid      */
      }

      /* Find frame for new _VECTOR_ture: must be large enough to hold
both _VECTOR_s... */
      UnionRect(&((**VECTOR_1).VECTOR_Frame),
            &((**VECTOR_2) VECTOR_Frame),&newFrame);
      (iv)

      drawPort = FrontWindow();
      if (VECTOR_ToBits(VECTOR_1,&theBits) == noErr) {
            (v)

            theVECTOR = Open_VECTOR (&newFrame);
      (vi)

            Draw_VECTOR (VECTOR_2,&((**VECTOR_2).VECTOR_Frame));
      (vii)
            CopyBits(&theBits, &(drawPort->portBits),
      (viii)
                &((**VECTOR_1).VECTOR_Frame),
                  &((**VECTOR_1).VECTOR_Frame),thePen,NULL);

            Draw_VECTOR (VECTOR_1,&((**VECTOR_1).VECTOR_Frame));
      (ix)
            Draw_VECTOR (VECTOR_2,&((**VECTOR_2).VECTOR_Frame));
      (x)

            Close_VECTOR();
      (xi)
            }

      return (the_VECTOR);
      (xii).
   }
```

Figure (2): Function Over_VECTOR. The function overlays two VECTOR data handles and returns a parsed VECTOR_Handle. The standard X-Windows procedure CopyBits is used to merge the two data handles to form a single contiguous overlay.

```
\**********************************************************************/
```

context) offscreen. Next, data handle VECTOR_1 and data handle VECTOR_2 are positioned on-screen. The onscreen position of the individual images is determined using the VECTOR_Frame rectangle in position (ix) and position (x). The Close_VECTOR procedure terminates the overlay process and the composite image is returned at position (xii). The function Over_VECTOR illustrates a number of important processes required to support parsed segmentation of VECTOR images: first, all data fields must be checked for validity prior to any overlay process. A single non-valid data structure prematurely terminates all operations. Second, all data procedures must have valid management of both

```
\*********************************************************************/

   void Global2_VECTOR (wPtr, thePt)
      WindowPtr   wPtr;
      Point             *thePt;
   /*  Convert global coordinates to 'VECTOR' coordinates. */
   {
            GrafPtr           savePort;

      if (!ValidPointer((Ptr)wPtr) || !ValidPointer((Ptr)thePt))
            (i)
            RETURN;

      GetPort(&savePort);
                  /* Save current port    */
      SetPort(wPtr);
      GlobalToLocal(thePt);
      (ii)
      Local2_VECTOR (wPtr,thePt);
      (iii)
      SetPort(savePort);
        /* Restore original port    */
   }
```

Figure (3): Function Global2_VECTOR. The function projects globally defined coordinates into a local coordinate system defined within a given image window.

```
\*********************************************************************/
```

radiometric data and associated geometry. Finally, each overlay can be transmitted to a parsed segmentation algorithm within a single data handle. Composite data fields are not required since pointer and handle arithmetic may be used to route both digital data and related attribute information. As shown in Figure (2), the VECTOR_Frame controls the position of a given overlay. As a result, the composite image (overlay or addition) varies with both the local and the global coordinates that are contained within the original VECTOR data. This

```
\**********************************************************************/

void Local2_VECTOR (wPtr, thePt)
      WindowPtr   wPtr;
      Point             *thePt;
/* Convert local coordinates to 'VECTOR' coordinates. */
/* Assumes that wPtr is the current port. */
{
      WData WD;

      if (!ValidPointer((Ptr)thePt))
            RETURN;

      GetWData(wPtr,&WD);
      if (!ValidWData(&WD))
      (i)
      /* Window has no WData record... */
            SetPt(thePt,-32767,-32767);
            /* Default error values */
      else {
            if (ValidHandle((Handle)(WD.vScrollBar)))
      (ii)
                  thePt->v += GetCtlValue(WD.vScrollBar);
            if (ValidHandle((Handle)(WD.hScrollBar)))
      (iii)
                  thePt->h += GetCtlValue(WD.hScrollBar);
      }
}
```

Figure (4): Function Local2_VECTOR. The function projects locally (within window) defined coordinates into VECTOR coordinate system.

```
\**********************************************************************/
```

problem is compounded when two or more images must be rectified to one another based upon dissimilar coordinates and/or projections. In Figure (3) a simple coordinate transform algorithm is provided to illustrate the general methods required for geometric projection of VECTOR data. In Figure (3), the Global2_VECTOR Procedure applies the ValidPointer function to check for data integrity prior to performing the geometric translation. Note that the ValidPointer function is applied at position (i) as opposed to the ValidHandle function since X-window data is stored within a pointer (whereas VECTOR raw image data is stored within a Handle). The Global2_VECTOR procedure performs the coordinate system translation using the standard GlobalToLocal data utility (position ii). The Local2_VECTOR procedure (shown in Figure 4) is used to re-project the digital information into VECTOR coordinates (i.e. formatted for import/export of VECTOR digital data). In Figure (4), the Local2_VECTOR function performs a number of necessary functions. First, the procedure checks the display window for data validity (position i). Next, the function applies the ValidHandle utility to check all associated controls within the window. The vertical controls are tested at position (ii), and the horizontal controls are tested at position (iii). If all tests are satisfactory, the Local2_VECTOR function returns a positive result. Note that the data structure is indexed if the result is satisfactory.

In Figure (5), the VECTOR_2Local procedure performs the converse operation to that described in Figure (4). In this new procedure, the original coordinates from the base VECTOR file are tested for validity and projected into the window coordinates of the newly registered image. As in Figure (4), the ValidPointer is used to test data validity prior to the baseline operation. Note that Figure (4) and Figure (5) are nearly identical in their operations. In the prior case (Figure 4), the transform is applied to the window data (noted &WD) shown at position (i). In the later case (Figure 5), the transform is applied to the VECTOR pointer data (noted &thePt) at position (i).

```
\*********************************************************************/
```

```
void VECTOR_2Local (wPtr,thePt)
        WindowPtr    wPtr;
        Point        *thePt;
        /* Converts '_VECTOR_ture' to local coordinates. */
        /* Assumes that _VECTOR_ture is associated with window W via its
WData record */

{
        WData        WD;

        if (!ValidPointer((Ptr)thePt))
        (i)
                RETURN;

        GetWData(wPtr,&thePt);
        if (!ValidWData(&thePt))
        /* Window has no WData record */
                SetPt(thePt,-32767,-32767);
        /* Default error values */
        else {
                if (ValidHandle((Handle)(thePt.vScrollBar)))
                        thePt->v -= GetCtlValue(thePt.vScrollBar);
                if (ValidHandle((Handle)(thePt.hScrollBar)))
                        thePt->h -= GetCtlValue(thePt.hScrollBar);
        }
}
```

Figure (5): Function VECTOR_2Local. The function projects VECTOR coordinates to locally defined window system.

```
\**********************************************************************/
```

7. VECTOR Import and Export

Image data is organized into a repeated scan-line sequence that requires parsed segmentation across geometric patterns. The function Read_VECTOR shown in Figure (6) performs this function using separate source buffers and separate destination buffers. As shown in Figure (6), the Read_VECTOR function relies upon many indexes to organize the parsed segmentation process. Initially, a source buffer of size 2*Blocksize is defined at position (i). The source buffer (of type XByte) is used to buffer all information between the source pointer (srcPtr at position ii) and the destination pointer (dstPtr at position ii). Note that the actual data is stored in an ancillary bitmap shown in position (iii). The ancillary bitmap (labeled theBits) contains a single row of VECTOR data, and is swapped into and out of memory as required to read/write the physical VECTOR data. In position (iv) and position (v) of Figure (6), the ValidPointer and the XTSTRLEN are respectively applied to test the VECTOR data for pointer (index) validity and string length validity. In position (vi), the XGetFile procedure is used to position a pointer to a given VECTOR file on the host platform. Note that the file type 'VECTOR' is defined between position (v) and position (vi) to include only VECTOR compatible images within the selection process. The process of unpacking the VECTOR image begins with declarations shown at position (vii). The number of lines in the file (nLines) is calculated based upon the ending line of the total VECTOR image. Note that since VECTOR data files are not of a fixed size, the number of lines to be unpacked must be calculated for each image in the data set. The unpacking process is managed by two separate pointers (srcPtr and dstPtr). Initially the variable srcPtr points to the initial data field within the source buffer (srcBuf). This process is shown in position (viii). In addition, the number of bytes required to reconstruct a new row of VECTOR data is defined within the variable dstBits.rowBytes. Note that the variable dstBits.rowBytes always begins with the number of bytes contained within a single row of VECTOR data. This declaration is shown in position (ix). As in the case of the source pointer, the destination pointer at position (x) always points to the initial data record within the buffer. In position (xi), the minimum bounding rectangle for the buffer

dstBits is defined. As shown, the buffer is of size [VECTORDepthBytes x 1]. In this regard, only one line of data is unpacked within the buffer at any given period. The read procedure formally begins with the
file position assignment shown in position (xii). In this assignment, the image data begins at position fStartReadPos within the base VECTOR file. The data is formally unpacked during the read process in paired blocks. Each block is of size BlockSize * 2 as shown in position (xiii). In position (xiv), the data is read from the VECTOR file and placed within the source buffer. If no error occurs during the read process (position xv), the index is set equal to the byte total of the data block (position xvi), and a single line of data is unpacked into the source buffer (position (xvii).

```
\*******************************************************************/

OSErr Read_VECTOR (theMap,theReply,startLine,endLine)

      BitMap          *theMap;
      SFReply     *theReply;
      int         startLine,endLine;
{

      SFTypeList  myFileTypes;
      BitMap            dstBits;
      Rect        showRect,lineRect;
      Point       sfOrigin;
      XByte       srcBuf[2*BlockSize];                      (i)
      Ptr         srcPtr,dstPtr;                            (ii)
      OSErr       errCode;
      long        fStartReadPos,count;
      int         fRefNum,nLines,i;
      char        theBits[VECTORRowBytes];
      (iii)

      if (!ValidPointer((Ptr)theMap) || !ValidPointer((Ptr)theReply) ||
      (iv)
                  (startLine >= endLine))
            return(nilHandleErr);

      if ((XTSTRLEN(theReply->fName) < 1) || !(theReply->good)) {
      (v)
            /* If reply.fName is <1 char long, call SFGetFile */
            myFileTypes[0] = 'VECTOR';
            SetPt(&sfOrigin,82,30);
            XGetFile(pass(sfOrigin),"Load which
file?",NULL,1,myFileTypes,     (vi)
                                  NULL,theReply);
            if ((XTSTRLEN(theReply->fName) < 1) || !(theReply->good))
                  return(abortErr);
      }
      nLines = endLine - startLine;                                   (vii)
      srcPtr = &(srcBuf[0]);                                          (viii)
      dstBits.rowBytes = VECTORRowBytes;                              (ix)

      dstBits.baseAddr = (Ptr)(&theBits[0]);
      (x)
      SetRect(&(dstBits.bounds),0,0, VECTORDepthBytes,1);
      (xi)        SetCursor(*(GetCursor(Xcursor)));

      errCode = FSOpen(theReply->fName,theReply->vRefNum,&fRefNum);
      if ((errCode == noErr) || (errCode == opWrErr)) {
            fStartReadPos = HeaderSize;
            errCode = SetFPos(fRefNum,fsFromStart,fStartReadPos);
      (xii)
      }
      if (errCode == noErr) {
            count = BlockSize * 2;                                    (xiii)
            if (errCode == noErr)
                  errCode = FSRead(fRefNum,&count,srcBuf);            (xiv)
      }
      if (errCode == noErr) {                                         (xv)
            count = BlockSize;                                        (xvi)
            SetRect(&lineRect,0,0,576,1);
            /* init rect bits are unpacked into */
            SetRect(&showRect,0,0,576,1);
            /* init rect bits copied from lineRect */
            srcPtr = (Ptr)srcBuf;                                     (xvii)
            dstPtr = dstBits.baseAddr;
            /* init pointer to unpacking bitfield*/

            for (i=0; (i<endLine) && (errCode == noErr); i++)
      (xviii)
            {
            /* For each line of bits,unpack  */

                  UnpackBits(&srcPtr,&dstPtr,_VECTORRowBytes);
      (xix)
                  dstPtr = dstBits.baseAddr;
                  /* reset pointer to unpacking bitMap */
                  if (i >= startLine) {
                        /* If line is within range, move to target... */

                        CopyBits(&dstBits,theMap,&lineRect,           (xx)
                              &showRect,srcCopy,NULL);
                        OffsetRect(&showRect,0,1);
                        /* walk target rect down one line */
                  }
                  if (((long)srcPtr - (long)srcBuf) > BlockSize) {
      (xxi)
                        /* Past first block of bytes, */
                        /* copy upper block onto lower block ...*/
                        BlockMove(&(srcBuf[BlockSize]),               (xxii)
                              &(srcBuf[0]),(Size)BlockSize);
                        /* Read next block into buffer... */
                        errCode =
FSRead(fRefNum,&count,&(srcBuf[BlockSize]));
                        srcPtr -= BlockSize;
                  }
                  if ((errCode == eofErr) && (count >= 0))
                        errCode = noErr;
            }
      }
      if (errCode == noErr)
            XTCloseFile(theReply);
      (xxiii)

      InitCursor();
      return(noErr);                                                (xxiv)
   }
```

Figure (6): Function Read_VECTOR. The function imports a single VECTOR compatible image file from a user defined source.

```
\*********************************************************************/
```

Data is unpacked a line at a time at position (xviii). Note that data is unpacked until the end of the file (endLine) is reached and provided no error has been encountered during the read process. The procedure UnpackBits at position (xix) is used to formally unpack the bits associated with each scanline and the expanded data is then positioned within the new vector file using the standard CopyBits utility (position xx). Note that the address of the source pointer and the address of the destination pointer must be indexed to correspond to the individual positions of each data record within the vector file. These calculations are performed in position (xxi). The standard BlockMove utility (position xxii) is then used to formally transfer the unpacked data into the window record. The function Read_VECTOR formally terminates when all lines of data are parsed and unpacked into the window record. When the data is completed, the file is closed (position xxiii) and a no error result is returned (position xxiv). The Read_VECTOR function illustrates a number of important features that are required to efficiently manage VECTOR data sets. First, all data records should be dynamically allocated to correspond to the exact size of the image volume. Since data sets vary in both width and depth, these parameters should not be statistically determined prior to the read process. Next, each data record should be unpacked using blocksize parameters. In this manner, the Read_VECTOR function uses only 2*BlockSize memory during the UnPack process and does not waste valuable resources. Finally, the Read_VECTOR function unpacks and sorts data using parsed segmentation methods. All indexes are determined in the course of the read/write process.

The Write_VECTOR function shown in Figure (7) is used to write Encapsulated Postscript (EPS) Binary compatible files. The function extracts information contained within a source buffer and systematically packages the data into VECTOR data structures. The function Write_VECTOR works with data that is stored within a contiguous XMap (shown in position i). The XMap is parsed into two separate buffers of size BlockSize (position ii). The source buffer srcBuf and the destination buffer dstBuf are required to respectively store a line of non-compressed and a line of compressed data. The non-compressed data is extracted from the XMap one line at a time, and is then compressed within the destination buffer dstBuf. The function Write_VECTOR opens a destination file at position (iii) and sets the initial file index at position (iv). In this example, the header record is primed with zeros at position (v) to indicate the actual location of the destination image. The formal packing procedure begins at position (vi). As shown, indexes are created at the current position of the data (position vii), the number of horizontal bytes in the image (hBytes at position viii), the number of vertical bytes in the image (vBytes at position ix), the vertical size of the resultant VECTOR file (vSize at position x), and the number of actual bytes placed within the data file (the variable goodBytes at position xi). At position (xii) a nested loop is declared for all bytes of data in the VECTOR image. Data is packed within each of the VECTORFileLines based upon the number of bytes (VECTORRowBytes) in each line. The VECTOR bytes are packed in 16 bit words with error checking for each of the odd and even bits in the word. The error checking is shown for the horizontal and vertical bytes in position (xiii). If an error is encountered during the bit packing operation, a zero value is placed within the resultant image (position xiv). This operation generally produces a white marker for most image processing and CAD systems. The source pointer and the destination pointer are exchanged in position (xv) and position (xvi) to allow room for a new line of VECTOR data. The data that is contained within the source pointer srcPtr is then packed using the standard PackBits utility (position xvii). The indexes for both the source pointer and the destination pointer are then re-calculated so that each buffer holds a new packet of VECTOR data. After the data exchange (position xviii), the information is written to the destination file (position xix) and the file size is incremented by the new byte count (position xx). The Write_VECTOR function formally terminates with the closing of the image file at position (xxi). The preceding functions SetEOF and FlushVol respectively set the end of file marker and clear the memory positions for the write process.

```
\*********************************************************************/

OSErr Write_VECTOR (theMap,theReply,creator,fileType)

      XMap        *theMap;                                         (i)
      SFReply     *theReply;
      OSType        creator,fileType;
{

      QDByte        srcBuf[BlockSize],dstBuf[BlockSize];
      (ii)
      QDByte        *curByte;
      Point         sfOrigin;
      OSErr         errCode;
      Ptr           srcPtr,dstPtr;
      long          dstBytes,fSize;
      int           fRefNum,hSize,vSize,hBytes,goodBytes,bit,i,j;

      if (!ValidPointer((Ptr)theMap) || !ValidPointer((Ptr)theReply))
            return(nilHandleErr);
      SetCursor(*(GetCursor(watchCursor)));

      errCode = FSOpen(theReply->fName,theReply->vRefNum,&fRefNum);
      (iii)
      if (errCode == fnfErr) {                          /* If non-
existant, then create it */
            errCode = Create(theReply->fName,theReply-
>vRefNum,creator,fileType);
            if (errCode == noErr)
            errCode = FSOpen(theReply->fName,theReply-
>vRefNum,&fRefNum);
      }
      if (errCode == opWrErr)
            errCode = SetFPos(fRefNum,fsFromStart,(long)0);
      (iv)
      if (errCode == noErr) {
            fSize = HeaderSize;
            for (i=0; i<HeaderSize; i++)
                  dstBuf[i] = 0;
      (v)
            errCode = FSWrite(fRefNum,&fSize,dstBuf);
      }
      if (errCode == noErr) {
      (vi)
            curByte = theMap->baseAddr;
      (vii)
            hBytes = theMap->rowBytes;
      (viii)
            hSize = theMap->bounds.right - theMap->bounds.left;
      (ix)
            vSize = theMap->bounds.bottom - theMap->bounds.top;
      (x)
            goodBytes = hSize / 8;
      (xi)

            for (j=0; (j<VECTORFileLines) && (errCode == noErr); j++) {
            (xii)
                  for (i=0; i<VECTORRowBytes; i++) {
                        if ((i < hBytes) && (j < vSize)) {
      (xiii)
                              srcBuf[i] = *curByte++;
                              if (i >= goodBytes)
                                    for (bit=15; bit>=0; bit--)
                                          if ((i*8) + (15-bit) > hSize)

      BitClr(&(srcBuf[i]),(long)bit);
```

```
                        }
                        else
                              srcBuf[i] = (QDByte)0;
      (xiv)
                  }
                  srcPtr = srcBuf;
      (xv)
                  dstPtr = dstBuf;
      (xvi)
                  PackBits(&srcPtr,&dstPtr,_VECTORRowBytes);
      (xvii)
                  dstBytes = (long)dstPtr - (long)dstBuf;
      (xviii)
                  errCode = FSWrite(fRefNum,&dstBytes,dstBuf);
      (xix)
                  fSize += dstBytes;
      (xx)
            }
      }
      if (errCode == noErr)
            errCode = SetEOF(fRefNum,fSize);
             /* total bytes including header    */
      if (errCode == noErr)
            errCode = FlushVol(NULL,theReply->vRefNum);

      XTCloseFile(theReply);
      (xxi)
      InitCursor();
      return(errCode);
}
```

Figure (7): Function Write_VECTOR. The function exports a single VECTOR compatible image file from a user defined source.

```
\*********************************************************************/
```

8. Conclusions

VECTOR digital data is stored in a variable field format that is efficiently managed using parsed segmentation algorithms. Parsed segmentation methods provide efficient access to all variable field data structures contained within the base image file. As shown in Figure (1) and Figure (2), vector images may be managed using single data handles that correspond to all fields and records within the data set. Using linear arithmetic, images can be added and overlayed using standard X11R6 libraries. The application of VECTOR data within CAD and GIS is illustrated using the geometric conversion utilities shown in Figure (3), Figure (4), and Figure (5). The conversion utilities provide efficient data translation methods for absolute position of the polygon data according to both local and global coordinate systems. While these procedures do not define an absolute projection algorithm, they illustrate efficient management of vector attribute information and global to local mapping of all primitives contained within a single vector image. In Figure (6) and Figure (7), import and export utilities are defined for the fast exchange of data information across standard Unix platforms. The utilities build upon earlier spooling methods defined within the ERO research program and may be used to efficiently manage large volumes of variable width data using segmented buffers. The segmented buffers are of variable width, depth, and size. Image data acquired in 8 to 32 bits can be projected using these algorithms. The projected data is compatible with Adobe (EPS) and AutoCad (DXF) data structures, and may be used to convert digital information from standard raster formats. While the enclosed algorithms do not form a comprehensive VECTOR library that is ultimately required to create complete data projections (from raster to vector), the procedures and functions illustrate how these data formats may be managed within a distributed operating environment.

9. References

Akl, S., Parallel Sorting, Academic Press, Orlando, Florida, 1985.

Anderson, James A., "Cognitive and psychological computation with neural models." in IEEE Transactions on systems, man and cybernetics, vSMC-13, no 5, Vol 2, 1994.

Baase, S., Computer Algorithm Design and Analysis, Addison Wesley, Reading, MA, 1992.

Baum, E.B. and D. Haussler, "What Size Net Gives Valid Generalization," Neural Computation, vol. 1, p151, 1989.

Beakley, G.W. and F.B Tuteur, "Distribution Free Pattern Verification Using Statistically Equivalent Blocks," IEEE Trans. Comput., vol. E-14, 1992.

Boardman, J. W., "Inversion of imaging spectrometer data using singular value decomposition", Proceedings IGARSS, 12th Canadian Symposium on Remote Sensing, v4, 1989.

Boardman, J. W., "Inversion of High Spectral Resolution Data", Proceedings SPIE, v1298, 1990.

Boardman, J. W., "Sedimentary Facies Analysis Using Imaging Spectrometry: A Geophysical Inverse Problem", Remote Sensing of the Environment, AVIRIS, 1992.

Boardman, J. W., "Spectral Angle Mapping: A Rapid Measure of Spectral Similarity", AVIRIS, 1993.

Chandrasekaran, B. and T.J. Harley, Comments "On the Mean Accuracy of Statistical Pattern Recognizers," IEEE Trans. Inform. Theory, Corresp., vol. IT-15 Rev 2, 1993.

Clark, R. N., Gallagher, A. J., and Swayze, G. A., "Material Absorption Band Depth Mapping of the Imaging Spectrometer Data Using a Complete Band Shape Least-Squares Fit with Library Reference Spectra", Proceeding of the Second Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) Workshop, JPL Publication 90-54, 1990.

Clark, R. N., Swayze, G. A., Gallagher, A. J., and Kruse, F. A., "Mapping with Imaging Spectrometer Data using the Complete Band Shape Least-Squares Algorithm Simultaneously Fit to Multiple Spectral Features", Proceeding of the Third Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) Workshop, JPL Publication 91-28, 1991.

Gibbons, A. and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, Cambridge, 1990.

Hornik, K., M. Stinchcombe and H. White, "Multilayer Feedforward Networks are Universal Approximators," Neural Networks 2, p359-366, 1989.

Karp, R., "Combinatorics, Complexity and Randomness," Communications of the ACM, vol. 29, no. 2, February 1986.

Kirkpatrick, S., C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," Science, vol. 220, p671-680, 1983.

Kruskal, C., "Searching, Merging and Sorting in Parallel Computation," IEEE Transactions on Computers, vol. c32, no. 10, October 1993.

Mazer, A.S., Martin M., Lee M., and Solomon, J. E., "Image Processing Software for Imaging Spectrometry Data Analysis", Remote Sensing of the Environment, v24, no 1, 1988.

Weigend, A.S., B.A. Huberman, and D.E. Rumelhart, "Predicting the Future: A Connectionist Approach," Submitted to the Journal of Neural Systems, April, 1990.

Yuhas, R. H., Goetz, A. F. H., and Boardman, J. W., "Discrimination Among Semi-Arid Landscape Endmembers using the Spectral Angle Mapper (SAM)

Algorithm", Summaries of the Third Airborne Geoscience Workshop, JPL Publication 92-14, v1, 1992.

Annex to

Fifth Report (07 May 1999)
Conditional Estimation of Vector Pattern in Remote Sensing and GIS

contract no. N 68171 97 9027
contractor: UvA,Applied Logic Laboratory/CCOM
Principal Investigator: Dr. M.Masuch

1. Statement showing amount of unused funds at the end of the covered period

2nd Incrementally Funded Period Total	$	
June 1999 - December 1999	$	50,000.00
3rd Incrementally Funded Period Total January 2000 - Augustus 2000	$	150,000.00
Total unused funds from January 1999 until August 2000	$	200,000.00

2. List of important property acquired with contract funds during this period

none